Introduction to Go

Raleigh Littles

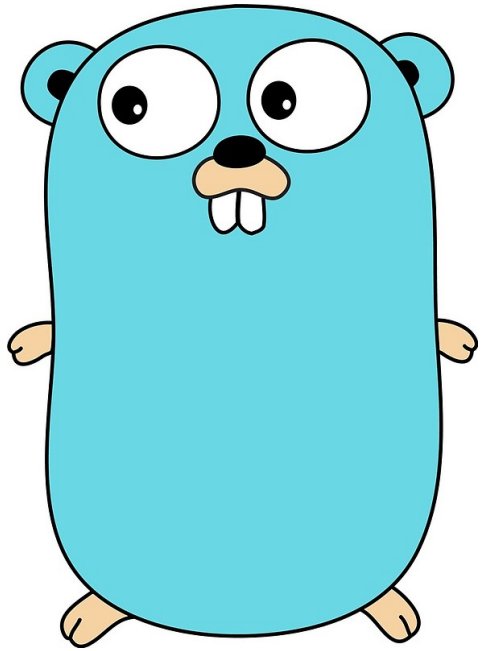
January 5, 2020

Overview

1 What is Go?

Go what?

- The simplest way to explain what Go is: C's cooler nephew*
- Like all cool things, it came out of Google
- Version 1.0 debuted March 2012



⚠Disclaimer: I don't actually know C ⚠

Where is Go being used anyways?

- Twitch: uses Go to handle chat [1]
- Uber: uses Go for running their geofence lookup service [2]
- Docker : LXC but better
- Kubernetes : Container management platform

And last but not least, the RDMC uses Go... (talk to Scott Johnson about this)

[1] https://blog.twitch.tv/en/2016/07/05/gos-march-to-low-latency-gc-a6fa96f06eb7/
[2] https://eng.uber.com/go-geofence/

The why

The primary reason Go exists is because Ken Thompson and Rob Pike (both of Unix fame!) hated C++. [3]

They wanted to make a language, that, in their exact words, was: [4]

1. comprehensible
2. statically typed
3. fast to work in
4. scales well
5. doesn't require tools, but supports them well
6. good at networking and multiprocessing

[0] https://www.drdobbs.com/open-source/interview-with-ken-thompson/229502480
[1] https://web.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf

C(omplicated)++

The biggest complaint that the Go creators had with C++ is it's complexity.
It's difficult to mathematically prove that one programming language is more complicated than the other, but one datapoint to look at is the number of reserved keywords.

Language	# of reserved words
C++	86
C	32
Python	31
Ruby	38
Java	51
Go	25

It looks like they have a point...

Devil's advocate

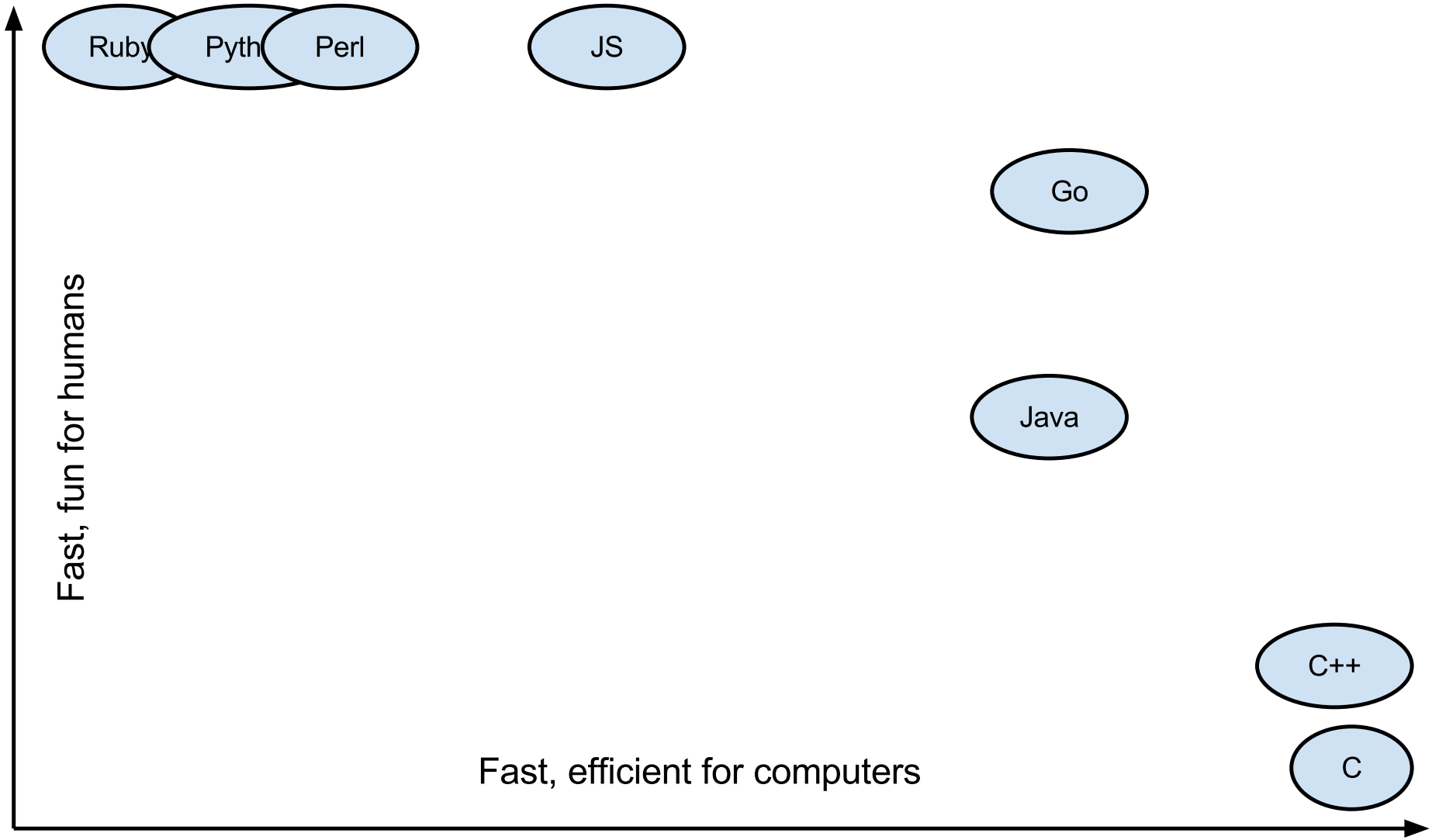
That being said, there are still some benefits to C++:

- It's fast as fuck
- It's type-safe

What if we could try to combine some of the benefits of C++ an easy-to-use language like Python or Ruby?

Can you see where this is ... *Go*ing?

This graph explains it well

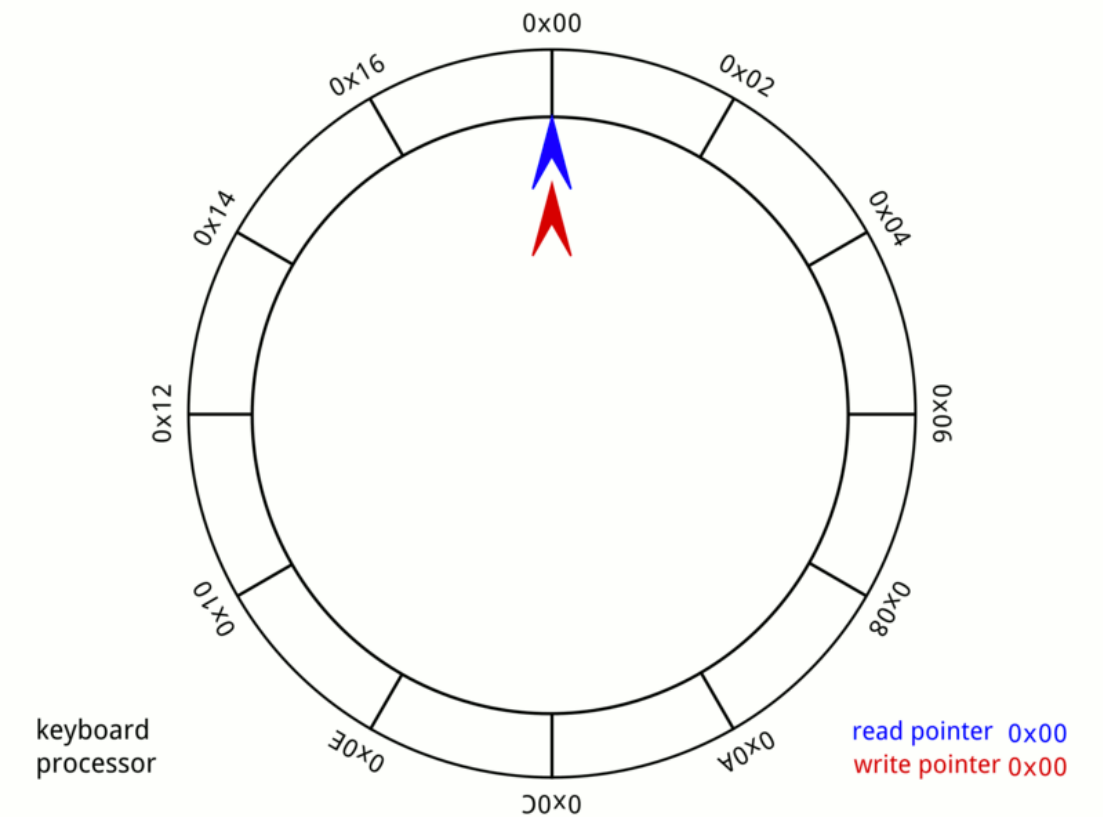


This is Rob Pike's opinion. I don't necessarily agree with it

Sike!

Normally this is where I'd show you an example of Hello World. But we're all adults here right?

Let's *Go* write ahead and implement a circular buffer :)



Implementation part 1

Let's start with the basics.

```
 1  package circular_buffer
 2
 3  import "errors"
 4
 5  type Buffer struct {
 6          arr []byte
 7          read_index int
 8          write_index int
 9          num_elements int
10  }
11
12  func NewBuffer(size int) *Buffer {
13          buf := Buffer{make([]byte, size), 0, 0, 0}
14          return &buf
15  }
```

Implementation part 2

Time to add read and write functionality.

```
16  func (b *Buffer) ReadByte() (result byte, err error) {
17          if (len(b.arr) == 0) || (b.num_elements == 0) {
18                  return 0, errors.New("Can't read from empty buffer!")
19          }
20
21          result = b.arr[b.read_index]
22          err = nil
23
24          b.read_index = (b.read_index + 1) % len(b.arr)
25          b.num_elements--
26
27          return
28  }
29
30  func (b *Buffer) WriteByte(c byte) (error) {
31          if (len(b.arr) == 0) {
32                  return errors.New("Can't write to an empty buffer")
33          }
34
35          if (len(b.arr) == b.num_elements) {
36                  return errors.New("Can't write to full buffer")
37          }
38
39          b.arr[b.write_index] = c
40          b.write_index = (b.write_index + 1) % len(b.arr)
41          b.num_elements++
42
43          return nil
44  }
```

Now what?

We've got code written. Now we have to talk about how to build and deploy our code!

Trust me, this is where the fun starts..

Honesty is the best policy

Thankfully, the other thing that Ken Thompson and Rob Pike didn't like about C++ was it's build system.

That statement is actually a lie, because **C++ doesn't have a build system**!!!!

Seriously though.

I mean sure, everyone just uses Make or CMake (if you're smart). But:

- I don't want to have to learn how to use Make or Cmake or Ninja or Autotools or any other build system
- I wanna focus on writing code for my application and my application alone

Go's build system

Go's promise is this:

Follow a handful of rules about how to organize your repository and you will never have to worry about makefiles again.

The rules are kind of annoying at first, tbh. You *will* curse a lot. But, in the end, the rules:

- make your repository more organized
- help people find code – no need for 15 different "commons" (I'm looking at you, modular)

Back to the circular buffer

On line 1, we declared our package's name to be `circular_buffer`.

One of the rules of Go is that each package must have its own directory.

Lets create a folder `cb` and place our file, called `circ_buffer.go` there.

Gucci Main()

Our application also needs an entrypoint! So we have to create a `main()` function.

```
1  // The contents of main.go
2  package main
3
4  // The relative directory where our
5  // circular buffer code lives
6  import ".\cb"
7
8  func main() {
9          circBuf := circular_buffer.NewBuffer(10)
10
11         // Do whatever else you want with your
12         // circular buffer here
13 }
```

Final structure

So let's look at what our structure looks like now:

```
|-- cb
|   |-- circ_buffer.go
|-- main.go
```

We're now ready to build our code. Literally just type in:
`$ go build`
That's it. Not even kidding. We have an executable ready to be executed!

Need to compile for a different architecture or OS? Simply set the `GOOS` and `GOARCH` environment variables during build time.

Still not impressed? Good

- A: *'Real world' applications are never that simple! What if I want to add more functions to my circular buffer implementation?*
- RL: Ok, so go add them to `circ_buffer.go`.
- A: *So, what if my Circular Buffer implementation needs 100 functions? I'm just supposed to put them in the same file? That's stupid I don't want to have an eight thousand line file!*
- RL: No need. You can create another file in that `cb` directory, add `package circular_buffer` to the first line, and add your new functions there.

- A: *Ok, that's fine.*

...

- A: *Hmm, now I have a problem – I have 100 files in the directory! It's hard for developers to find my code.*
- RL: Why on earth do you have 100 files for a circular buffer implementation?
- A: *My circular buffer implementation only needs 75 files, but there are about 25 files that contain my helper functions, duh!*
- RL: That's ... helpful. Try creating another directory, and a new package, that's called 'helper_functions' – then if you need to use them in your actual circular buffer code, you can simply do `import circular_buffer/helper_functions`. Although... keep in mind, Go will not let you create circular dependencies in packages – this is for your own benefit!

But seriously. Follow the freaking layout guidelines, and usually you can just get by with `go build` alone.
The one caveat is that when dealing with 3rd party dependencies, there is a more advanced system called Go Modules... not really going to get into that right now (but trust me, it's not as painful as Makefiles).
TODO: Include picture of RDMCD Go module file here

Remember bullet point #6?

Let's talk about concurrency a little bit.
One of the big design goals of Go was to make creating concurrent programs simple.
It's certainly possible to write concurrent/multithreaded code in C++ relatively simply:

```
1   #include <thread>
2   #include <iostream>
3   using namespace std;
4
5   void func() {
6           cout << "Inside of func() " << endl;
7   }
8
9   int main() {
10          thread th(&func);
11          // th.join();
12          cout << "Outside thread" << endl;
13  }
```

But can we do better?

Well yes and no

For an example that straightforward, not really. The equivalent Go code would look like:

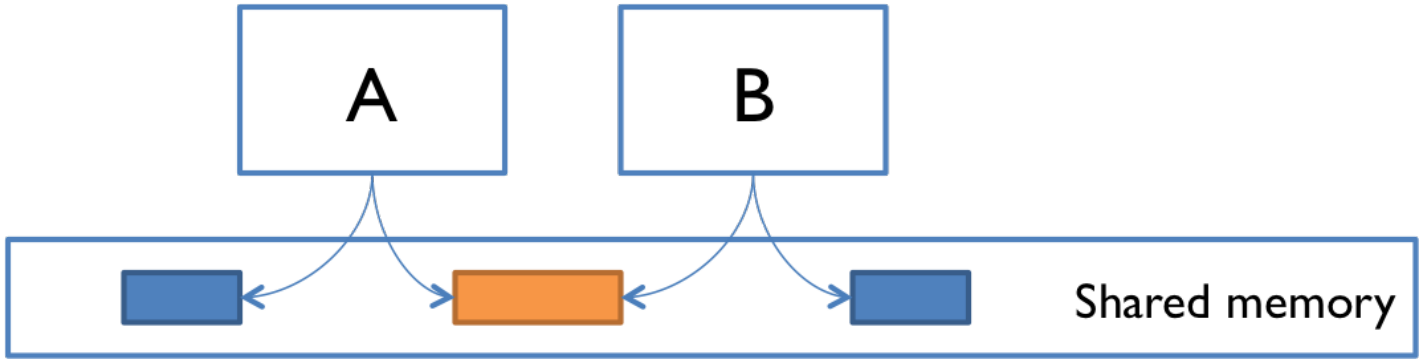
```
 1  package main
 2  import "fmt"
 3
 4  func fun() {
 5          fmt.Println("Inside thread!")
 6  }
 7
 8  func main() {
 9          fmt.Println("Main thread")
10          go fun()
11  }
```

But again, are real-world concurrency examples really ever that simple?

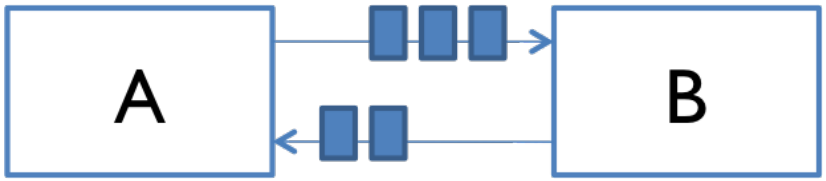
Anyone remember cable?

Generally speaking, the two types of concurrency programming models are:

1. Shared memory



2. Message passing



Go uses the second approach, and to accomplish this it uses something called **channels**

Channels

Channels are literally just, well, channels, between two goroutines – things go in one end of the channel and out the other end.
The syntax for channels is actually clever:

```
1  // Create a channel that holds strings
2  messages := make(chan string)
3
4  // put 'ping' into the channel
5  // (this typically happens in another thread)
6  messages <- "ping"
7
8  // read one message out of the channel
9  msg := <- messages
```

To be fair, you can use channels in C++ using Boost:
`https://www.boost.org/doc/libs/1_65_0/libs/fiber/doc/html/fiber/synchronization/channels.html`

(Is there anything that *isn't* in Boost?)

Try not to aim for your foot

In C++, to achieve *safe*, concurrent code in applications that share data, you use:

- atomic types (see std::atomic) that have been guaranteed to be thread-safe
- mutexes to synchronize access to some resource

Mess up on either of these and you're gonna have a bad time.

Meanwhile

Channels are inherently thread-safe in their implementation – meaning data races aren't possible.
That being said, you do still have to worry about the traditional concurrency bugs.. deadlock, for example.
There are still some specific rules about how to use channels that you have to follow (e.g. not reading from an empty channel).
Also, if using a channel is overkill, you still *can* use a traditional mutex like you would in C++.

Odds and Ends

We're going to switch gears a little bit to talk about a few more miscellaneous Go topics that didn't really fit in earlier.
They are listed in decreasing order of importance:

- Generics
- Interfaces
- Slices

"lol no generics"

Right now, the single biggest criticism of Go is that it lacks generics. The simplest possible definition of "generics" is basically: writing code for a type but you don't know what that type is right now.

The problem this creates is that if you have a function `f` that works on an integer parameter, you have to re-write it a second time to work with strings.

Or do you?...

This sounds really similar to templates, doesn't it? But.. templates (the way C++ does it) is actually an extension of the idea of 'generics', and the way they work under the hood is actually different than how generics work in a language for a Java for example. But that's a topic for a different day..

Interfaces

Well, actually, yeah you kind of do... BUT, there's one slight workaround called 'the empty interface'.
First let's talk about what a regular interface looks like in Go:

```
1  type polygon interface {
2          area() float64
3          perimeter() float64
4  }
```

We can now define our own structs, implement these methods, and then we can pass those objects to methods that take our interface as a parameter type:

```
1  func printMeasurements(p polygon) {
2      fmt.Println(p.area())
3      fmt.Println(p.perimeter())
4  }
```

Empty interface

When you use an interface type as a function parameter, it tells the compiler: 'whatever type I pass into this function, must at least implement the methods of that interface'.

But the empty interface (denoted as `interface{}`), by definition, has no methods in it, so any possible type can be substituted for that interface.

Consider a rollercoaster ride at an amusement park. The sign next to the ride says: *you must be at least 48 inches tall to ride*.

If the sign instead said: *you must be at least 0 inches tall to ride* – would there be anyone who **couldn't** go on that ride?

roll Call

This concept should be familiar to C developers, who have been using `void*` similarly since the first World War.

```
1  int a = 1;
2  float b = 2.3;
3  void *p;
4  p = &a;
5  printf("Integer variable is = %d", *( (int*) p) );
6  p = &b;
7  printf("\nFloat variable is = %f", *( (float*) p) );
```

Mathematically, this concept is called a *top type**, and it exists in other common programming languages as well.

Slices

Go, like any other sane language, has arrays. These are your pretty boring, fixed-size, same-type arrays, that you'd find in C.

```
1  primes := [6]int{2, 3, 5, 7, 11, 13}
```

Slices, on the other hand, get used a bit more often. These are like dynamically-sized arrays, and are much easier to work with than slices. You can create them through slicing an existing array or even just creating a slice directly:

```
1   // One way to create a slice
2   primes := []int{2, 3, 5, 7, 11, 13, 17}
3
4   // Appending to a slice
5   primes = append(primes, 19)
6
7   // One way to initialize a slice
8   sliceOfAllZeros := make([]int, 5)
9
10  // Slice-ception
11  var s []int = primes[1:4]
```

So yeah. We've covered basically all the major topics in Go, and at this point you're pretty much ready to begin reading and writing basic Go code. The only things worth mentioning that I didn't get to talk about are:

- The `for` loop syntax
- `select` statement
- `defer` statement
- The string system, e.g. strings, runes, bytes, etc.

For those interesting in learning more, I recommend you check out A Tour of Go: `https://tour.golang.org/`